Implementation of Huffman Coding as Encryption Algorithm Analysis

Ahmad Farid Mudrika - 13522008[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13522008@itb.ac.id

***Abstract*—Huffman encoding stands as a cornerstone in file compression, providing an efficient means to reduce data size. This essay introduces a program designed to encode files using a Huffman encoding tree derived from a base file. The implementation incorporates binary trees and encryption techniques, ensuring both compression and data security. The essay delves into the intricacies of Huffman encoding, exploring variable-length codes, codebook generation, and adaptive coding. It addresses decoding algorithms to ensure successful retrieval of the original data.**

***Keywords*—Binary tree, Compression, Encryption, Huffman encoding,**

I. INTRODUCTION

In this nonstop era of digitalization, where everything is turning digital, the quest for secure data transmission has become increasingly crucial. While securing the transmission itself is important, we can't ignore the possibility of hijacking. One of the way to secure the data itself is through encryption.

There have been many encryption algorithms explored. In this paper, we will explore using Huffman encoding as an encryption algorithm. Unlike traditional methods, which often involve complex mathematical operations, Huffman encoding presents a unique and innovative approach to data security. The proposed method involves taking a base file, constructing the Huffman tree based on its contents, and subsequently encoding other files using the established tree structure. This approach not only adds an extra layer of security to the data but also contributes to the realm of encryption by integrating a well-established compression technique.

The synergy between Huffman encoding and encryption aims to provide a comprehensive solution that not only safeguards data integrity but also optimizes storage space, acknowledging the intertwined challenges of security and efficiency in the digital landscape.

II. THEORETICAL BASIS

A. Binary Tree

A binary tree is a fundamental data structure in computer science and mathematics that represents a hierarchical organization of data in a graphical form. It falls under the broader category of trees, which are acyclic, connected graphs. The structure of a tree is hierarchical, resembling an inverted tree with a single root node from which branches extend downward. Each branching point in the tree is referred to as a node, and the connections between nodes are known as branches.

In a binary tree, each node can have at most two children, known as the left child and the right child. The nodes connected to these children are considered to be in a particular structural direction, with the left child appearing to the left and the right child appearing to the right. The unique characteristics of binary trees make them particularly well-suited for various applications in computer science, such as efficient searching algorithms, data storage, and expression parsing.

The terminal nodes in a binary tree, those without any children, are called leaves. These leaves are the endpoints of the branches and represent the smallest units of the tree structure. Importantly, every child in a binary tree is also a binary tree itself, allowing for a recursive and hierarchical representation of data.

Binary trees find applications in diverse domains, including database structures, algorithm design, and syntax trees in parsing. Their inherent hierarchical nature makes them useful for representing relationships and dependencies between different entities in a way that facilitates efficient traversal and manipulation of the underlying data.

The structural simplicity of binary trees, with their clear left and right directional references, contributes to their versatility in algorithmic design. Algorithms such as binary search, which leverages the ordered structure of binary search trees, demonstrate the practical significance of binary trees in achieving efficient search operations.

In summary, binary trees provide a flexible and hierarchical structure for organizing data in computer science. Their characteristics, including nodes, branches, leaves, and the binary nature of children, make them essential in various computational tasks, contributing to the development of efficient algorithms and data structures.[1]

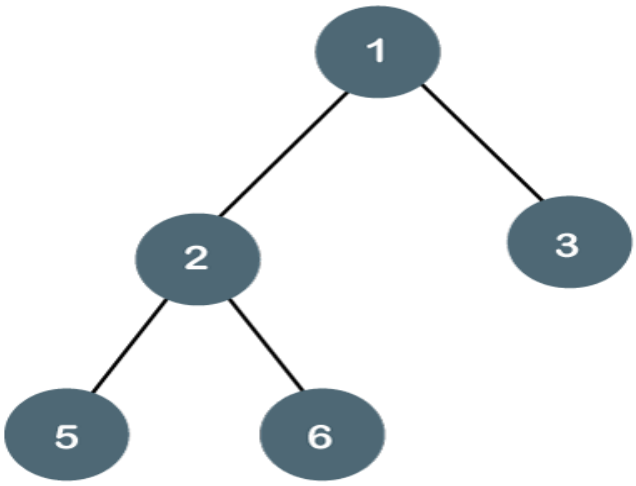

Figure 2.1 Binary Tree Illustration
https://www.javatpoint.com/binary-tree

B. Huffman Encoding

Huffman Encoding is a technique to compress data to reduce its size without any loss in the details. It was first developed by David Huffman. It uses binary tree as a media to compress the data.

Huffman encoding can be done through several steps, which are:[1]

1. Calculate the appearing frequency of each symbol in a string.

Figure 2.2 Huffman encoding tutorial
https://www.programiz.com/dsa/huffman-coding

2. Sort the character in descending or ascending order. This only serve to simplify the process, you don't actually need to do it. In the example below, we sort it in an ascending order.

Figure 2.3 Huffman encoding tutorial
https://www.programiz.com/dsa/huffman-coding

3. Make a leaf node of each symbol.
4. Take out two symbol with the least frequency, make a binary tree containing them. Whether the lesser frequency is in the left or right node doesn't matter, as long as we are consistent with it. Store back the tree in the symbol list, with the frequency being the sum of the two's frequency.

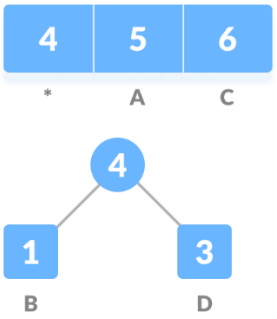

Figure 2.4 Huffman encoding tutorial
https://www.programiz.com/dsa/huffman-coding

5. Repeat step 4 until you are left with a one-element list and a tree containing all of the symbol. This tree is called the Huffman tree.

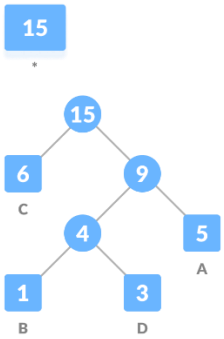

Figure 2.5 Huffman encoding tutorial
https://www.programiz.com/dsa/huffman-coding

6. Assign 0 or 1 on each edges of the tree. The order doesn't matter, as long as you do it consistently. In this example, I assign 0 on the left edge, and 1 on the right.

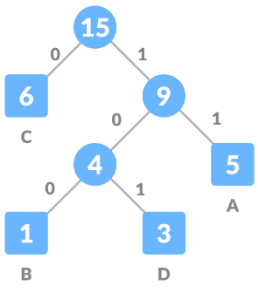

Figure 2.6 Huffman encoding tutorial
https://www.programiz.com/dsa/huffman-coding

7. Each symbol is then recognized by a code. This code is the edges taken to reach them from the root tree.

Character	Frequency	Code
A	5	11
B	1	100
C	6	0
D	3	101

Figure 2.7 Code table of Huffman encoding
https://www.programiz.com/dsa/huffman-coding

III. PROGRAM IMPLEMENTATION

I use the code from https://www.javatpoint.com/huffman-coding-using-python as a base and then change it to fit my needs. The complete code is at my github, https://github.com/frdmmm/Huffman-encoding.git.

A. Nodes Class

```python
class Nodes:
    def __init__(self, probability, symbol, left = None, right = None):
        self.probability = probability
        self.symbol = symbol
        self.left = left
        self.right = right
        self.code = ''
```

Figure 3.1 Node Class
Private Documentation

The Nodes class contains multiple elements, which are:

1. Probability
   This stores the probability of the symbol appearing in the string (frequency/string length).
2. Symbol
   This stores the symbol represented by the node.
3. Left
   This stores other node that is the left child of the node.
4. Right
   This stores other node that is the right child of the node.
5. Code
   This stores the code (in binary) that represents the symbol.

B. Calculate Probability Function

```python
def CalculateFrequency(the_data):
    symbols = dict()
    for item in the_data:
        if symbols.get(item) == None:
            symbols[item] = 1
        else:
            symbols[item] += 1
    return symbols
```

Figure 3.2 Calculate Probability Function
(Source : Private Documentation)

This function calculates the frequency of a symbol appearing in the string. It then returns a dictionary with the key being the symbol, and the value is the frequency of that symbol appearing in the string.

C. CalculateCodes Function

```python
the_codes = dict()

def CalculateCodes(node, value = ''):
    # a huffman code for current node
    newValue = value + str(node.code)

    if(node.left):
        CalculateCodes(node.left, newValue)
    if(node.right):
        CalculateCodes(node.right, newValue)

    if(not node.left and not node.right):
        the_codes[node.symbol] = newValue

    return the_codes
```

Figure 3.3 CalculateCodes Function
(Source : Private Documentation)

This function recursively call itself until the node parameter is a leaf (a tree without a child), and then added the symbol of the current node and its code to the external dictionary variable the_codes, which it then returned.

D. OutputEncoded Function

```python
def OutputEncoded(the_data, coding):
    encodingOutput = []
    for element in the_data:
        # print(coding[element], end = '')
        encodingOutput.append(coding[element])

    the_string = ''.join([str(item) for item in encodingOutput])
    return the_string
```

Figure 3.4 OutputEncoded Function
(Source : Private Documentation)

This function takes a parameter called coding, which is a dictionary with symbol as key and the code as a value, and the_data, which is the string that we want to encode.

E. HuffmanEncoding Function

```python
def HuffmanEncoding(the_data):
    symbolWithFreqs = CalculateFrequency(the_data)
    symbols = symbolWithFreqs.keys()
    the_nodes = []
    for symbol in symbols:
        the_nodes.append(Nodes(symbolWithFreqs.get(symbol), symbol))
    while len(the_nodes) > 1:
        # sort based on frequency
        the_nodes = sorted(the_nodes, key = lambda x: x.frequency)
        right = the_nodes[0]
        left = the_nodes[1]
        left.code = 0
        right.code = 1
        newNode = Nodes(left.frequency + right.frequency, left.symbol + right.symbol, left,
right)
        the_nodes.remove(left)
        the_nodes.remove(right)
        the_nodes.append(newNode)
    huffmanEncoding = CalculateCodes(the_nodes[0])
    # print("symbols with codes", huffmanEncoding)
    encodedOutput = OutputEncoded(the_data,huffmanEncoding)
    return encodedOutput, the_nodes[0]
```

Figure 3.5 HuffmanEncoding Function
(Source : Private Documentation)

This function wraps the previous functions and returned encodedoutput, which is the string(the_data) which have been encoded using Huffman Encoding, and the_nodes[0], which is the root of the tree. First, this function calculates the frequency of each symbol in the_data string, make a list of nodes, which

store all the leaf of the tree. It then does a loop until there is only one element left in the list of nodes(the_nodes), in which it create take out the two symbol with the least frequencies, make a tree(node) with it, and put the resulted node back into the list of nodes. It then get the code of each symbol in the tree in a dictionary called huffmanEncoding using CalculateCodes function, get the encoded string using OutputEncoded function, and returns it along with the tree.

F. HuffmanEncodingWithBaseTree Function

```
def HuffmanEncodingWithBaseTree(the_data, base_tree):
    the_codes=CalculateCodes(base_tree)
    encodedOutput = OutputEncoded(the_data, the_codes)
    return encodedOutput
```

Figure 3.6 HuffmanEncodingWithBaseTree Function
(Source : Private Documentation)

This function takes a parameter called the_data, which is the string to be encoded, and base_tree, which is the tree to base the encoding on. This then return the encodedOutput.

G. HuffmanDecoding Function

```
def HuffmanDecoding(encodedData, huffmanTree):
    Root = huffmanTree
    decodedOutput = []
    for x in encodedData:
        if x == '1':
            huffmanTree = huffmanTree.right
        elif x == '0':
            huffmanTree = huffmanTree.left
        try:
            if huffmanTree.left.symbol == None and huffmanTree.right.symbol == None:
                pass
        except AttributeError:
            decodedOutput.append(huffmanTree.symbol)
            huffmanTree = Root

    string = ''.join([str(item) for item in decodedOutput])
    return string
```

Figure 3.7 HuffmanDecoding Function
(Source : Private Documentation)

This function decode the encodedData variable using huffmanTree variable as a base.

IV. ANALYSIS

We can use the program by calling python main.py with several arguments, which are:

1. Basefile.
2. Encode or decode (e or d).
3. File to decode or encode.

The following test case use this as base.txt

```
base.txt
` 1 2 3 4 5 6 7 8 9 0 - =
q w e r t y u i o p [ ] \
a s d f g h j k l ; '
z x c v b n m , . /

~ ! @ # $ % ^ & * ( ) _ +
Q W E R T Y U I O P { } |
A S D F G H J K L : "
Z X C V B N M < > ?
```

Figure 4.1 base.txt
(Source : Private Documentation)

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed do eiusmod tempor incididunt ut
labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco
laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat
non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed do eiusmod tempor incididunt ut
labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco
laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat
non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed do eiusmod tempor incididunt ut
labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco
laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat
non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
```

Figure 4.2 s.txt
(Source : Private Documentation)

```
Base tree symbol dictionary:  {'I': '00000000', 'U': '00000001', 'Y': '00
000010', 'T': '00000011', 'R': '00000100', 'E': '00000101', 'W': '0000011
0', 'Q': '00000111', '+': '00001000', '_': '00001001', ')': '00001010', '
(': '00001011', '*': '00001100', '&': '00001101', '^': '00001110', '%': '
00001111', '$': '00010000', '#': '00010001', '@': '00010010', '!': '00010
011', '~': '00010100', '/': '00010101', '.': '00010110', ',': '00010111',
 'm': '00011000', 'n': '00011001', 'b': '00011010', 'v': '00011011', 'c':
 '00011100', 'x': '00011101', 'z': '00011110', "'": '00011111', ';': '001
00000', 'l': '00100001', 'k': '00100010', 'j': '00100011', 'h': '00100100
', 'g': '00100101', 'f': '00100110', 'd': '00100111', 's': '00101000', 'a
': '00101001', '\\': '00101010', ']': '00101011', '[': '00101100', 'p': '
00101101', 'o': '00101110', 'i': '00101111', 'u': '00110000', 'y': '00110
001', 't': '00110010', 'r': '00110011', 'e': '00110100', 'w': '00110101',
 'q': '00110110', '=': '00110111', '-': '00111000', '0': '00111001', '9':
 '00111010', '8': '00111011', '7': '00111100', '6': '00111101', '5': '001
11110', '4': '00111111', '\n': '01000', '3': '01001000', '2': '01001001',
 '1': '01001010', '`': '01001011', '?': '0100110', '>': '0100111', '<': '
0101000', 'M': '0101001', 'N': '0101010', 'B': '0101011', 'V': '0101100',
 'C': '0101101', 'X': '0101110', 'Z': '0101111', '"': '0110000', ':': '01
10001', 'L': '0110010', 'K': '0110011', 'J': '0110100', 'H': '0110101', '
G': '0110110', 'F': '0110111', 'D': '0111000', 'S': '0111001', 'A': '0111
010', '|': '0111011', '}': '0111100', '{': '0111101', 'P': '0111110', 'O'
: '0111111', ' ': '1'}
```

Figure 4.3 Base Tree symbol dictionary
(Source : Private Documentation)

Figure 4.4 A part of encoded s.txt
(Source : Private Documentation)

```
Enter the file name to store the encoded data: test1
```

Figure 4.5 File to store the encoded data
(Source : Private Documentation)

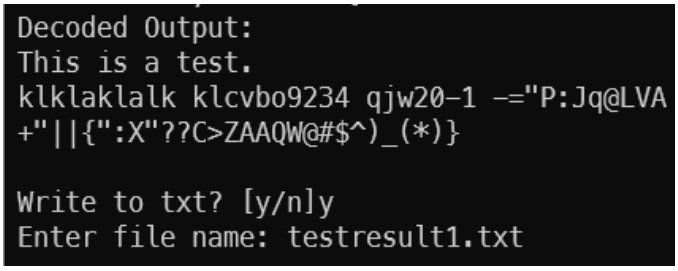

Figure 4.6 Decoding test1
(Source : Private Documentation)

You can store the encoded data as a txt file, but I don't recommend it. That is because when you store it as a txt file, it will treat every single binary in the encoded file as a character, and with each character being 8 bit long in the dictionary, it will result in 8 times size increasement.

If we take a look at Fig.4.3, we see that most of the character is represented as an 8 bit binary. This means that we will not see any significant compression ratio. We can try to combat this by making each character appearance frequency more diverse, or making a base.txt that suits our need more.

If we change the base.txt to diversify the frequency of each character, it will result in an overall shorter encoded file.

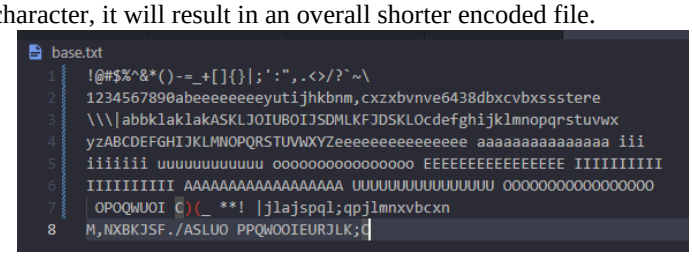

Figure 4.7 New base.txt
(Source : Private Documentation)

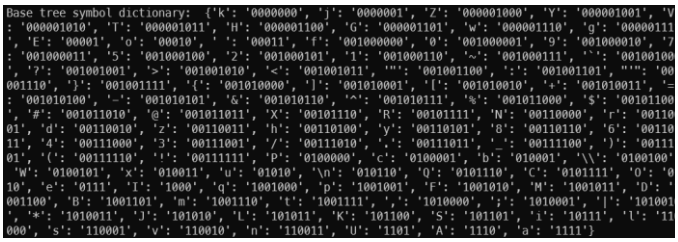

Figure 4.8 New base tree symbol dictionary
(Source : Private Documentation)

The size of the binary with the first base.txt is 10kb, and the size after the change is 8kb. So, we can still get the compression aspect of Huffman Coding. We just need to adjust our base.txt.

V. EVALUATION

Here are several pros and cons of using Huffman Coding as an encryption algorithm.

Pros:

1. Compression Efficiency
   Huffman coding excels at compressing data, representing frequently occurring symbols with shorter codes. This can lead to efficient use of storage space and reduced transmission times.
2. Simple Implementation
   The implementation of Huffman coding is relatively simple and easy to understand, making it accessible for educational purposes and practical applications where simplicity is a priority.
3. Key-based Encryption
   By using base.txt to generate the Huffman tree, we get a key-based element to the encoding process. This adds a layer of security, as the base.txt file is required in encoding and decoding.
4. Easily Customizable Key
   In traditional Huffman coding, the construction of the Huffman tree is solely based on the frequency distribution of symbols in the input data. However, by incorporating a key.txt file, the encoding process becomes more flexible and adaptable.

Cons:

1. Limited Security
   While Huffman coding with a base.txt file adds an additional layer of security, it is important to note that it may not provide the same level of security and features as dedicated encryption algorithms designed for confidentiality.
2. No Encryption of Key.txt
   Because the key.txt file is not itself encrypted or protected, an attacker gaining access to the key could potentially compromise the security of the encoded data. So, we will need a very secure means of transmission to send the base.txt.
3. Static Key
   The base.txt file, once created, remains static unless explicitly changed. This lack of dynamic adaptability can be a limitation in scenarios where regular key changes are desired for enhanced security.
4. No Integrity Verification
   Huffman coding, by itself, does not provide integrity verification. If an attacker tampers with the encoded data, there is no built-in mechanism to detect or correct such tampering.

VI. CONCLUSION

In conclusion, while Huffman coding may not be inherently designed as an encryption algorithm, its unique characteristics make it a viable option for encryption algorithm for small scale, educational and personal purposes. The algorithm's efficiency in data compression, customizable key management, and simplicity of implementation contribute to its suitability for educational settings where learners can gain hands-on experience with encoding and decoding processes.

It should be noted, however, that Huffman coding have limitations in terms of security when compared to dedicated encryption algorithms. While it may serve educational and personal needs effectively, for applications demanding robust confidentiality and integrity, exploring established cryptographic methods remains the recommended approach.

VII. ACKNOWLEDGMENT

The author would like to express gratitude to several parties for their contributions to this paper. First and foremost, sincere thanks are extended to God for providing guidance throughout the process of learning and writing. Additionally, the author acknowledges the invaluable support and teachings received from the lecturer of ITB Discrete Mathematics IF2120, Mrs. Nur Ulfa Maulidevi. Her knowledge and guidance have significantly enriched the learning experience in the class. Special thanks are also extended to the author's family and friends for their unwavering support throughout the entire semester.

REFERENCES

[1] https://www.scaler.com/topics/data-structures/binary-tree-in-data-structure/ accessed at December, 9 2023.

[2] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/23-Pohon-Bag2-2023.pdf accessed at December 7 2023.

[3] https://www.javatpoint.com/huffman-coding-using-python accessed at December 6, 2023

[4] https://www.programiz.com/dsa/huffman-coding accessed at December 8, 2023.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 11 Desember 2023

Ahmad Farid Mudrika 13522008